MP 2 — Higher Order Functions and User-Defined Types

CS 440 – Fall 2006

Revision 1.1

Assigned September 20, 2006
Due September 29, 2006
Extension October 1, 2006

1 Objectives and Background

This MP is designed to give you experience in using the higher order functions discussed in class. You will also use some datatypes to model some nice data-structures, and at the end, you will write a simple interpreter.

2 Problems

To warm up, write down the definitions of `fold_right`, `map`, and `zip`. You can use these in your solutions, or use the `List` library. To use the library version, prepend "`List.`" at the beginning, e.g., `List.map`.

2.1 Writing Higher Order Functions

1. Write the function `filter f xx : ('a -> bool) -> 'a list -> 'a list` which takes a predicate `f` and a list `xx`, and return the list of all the elements of `xx` which satisfy `f`. Use recursion.

2. Write the function `forall f xx : ('a -> bool) -> 'a list -> bool` which takes a predicate `f` and a list `xx`, and return `true` if all of the elements of `xx` satisfy `f`. Use recursion.

3. Write the function `exists f xx : ('a -> bool) -> 'a list -> bool` which takes a predicate `f` and a list `xx`, and return `true` if one or more of the elements of `xx` satisfy `f`. Use recursion.

4. Write the function `multimap ff xx : ('a -> 'a) list -> 'a list -> 'a list` which takes a list of functions `ff` and maps each of them to the list `xx`. Note that $multimap\ [f_0; f_1; \cdots; f_n]\ xx \equiv map\ f_0\ (map\ f_1\ (\cdots map\ f_n\ xx) \cdots)$. Use recursion.

2.2 Writing Higher Order Functions, II

5. Write the function `filter2 f xx : ('a -> bool) -> 'a list -> 'a list`, but this time do NOT use recursion.

6. Write the function `forall2 f xx : ('a -> bool) -> 'a list -> bool`, but this time do NOT use recursion.

7. Write the function `exists2 f xx : ('a -> bool) -> 'a list -> bool`, but this time do NOT use recursion.

8. Write the function `multimap2 ff xx : ('a -> 'a) list -> 'a list -> 'a list`, but this time do NOT use recursion. (The body of this function can be written in only two tokens! Your solution can be longer. Hint: $fun\ x\ \rightarrow\ f\ x\ \equiv\ f$.)

2.3 Trees

Most programming languages will be stored internally as different kinds of trees. These problems will help you get used to some of the techniques.

For the following problems, use this type:

```
type bst = Branch of int * bst * bst
         | Empty
```

A "leaf" will be a branch with two `Empty` children.

Here's a nice utility you can use.

```
let rec showbst t =
  match t with
  | Empty -> "x"
  | Branch (i,left,right) -> "(" ^ (string_of_int i) ^ " " ^
                             (showbst left)  ^ " " ^ (showbst right) ^
                             ")"
```

9. Write the function `add t i : bst -> int -> bst`. If the element is already in the tree, return the tree unchanged.

For the next problems it's useful to have some sample trees around. Contemplate the following code:

```
let t1 = List.fold_right (fun a b -> add b a)
                         (List.reverse [4;2;6;1;3;5;7])
                         Empty;;
```

10. Write the function `find t i : bst -> int -> bool`.

11. Write the function `delete t i : bst -> int -> bst`.

12. Write the function `mapbst f t : (int -> int) -> bst -> bst`.

13. Write the function `foldbst f t z : (int -> 'a -> 'a -> 'a) -> bst -> 'a`. The function `f` will take three arguments: the integer of the branch, the result of a left recursion, and the result of a right recursion. The `z` value is what should get returned when an `Empty` is encountered.

14. Using `mapbst`, write a function `incbst` which increments all the elements of the tree.

15. Using `foldbst`, write the function `sumbst t : bst -> int`, which takes the sum of a bst.

16. Using `foldbst`, rewrite the `showbst` function (call it `showbst2`).

17. Using `foldbst`, write a function `isbst t : bst -> bool`, which checks to see if `t` is in fact a legal binary search tree.

    Hint: You will find this easier if you declare your own type to keep track of the recursive results, and then make a wrapper function to convert that result into a boolean. There are three kinds of recursive results: empties, ranges (i.e., the subtree has values from $a \ldots b$), and invalid.

    If you get stuck, try writing it recursively first.

2.4 Interpreter

Here is a type for a calculator language.

```
type calc = Int of int
          | BinOp of string * calc * calc
          | UnOp of string * calc
```

18. Using the above type, write a calculator function `interp : calc -> int`. It should handle plus, minus, times, and unary minus. You do not have to check for invalid inputs, but you may extend the calculator all you like. In the next MP, you will use an environment to handle variables.

    Hint: it's easier to test if you save your programs to variables. For example, to save the expression $2 \times 3 + 5$, write something like:

    ```
    let p1 = BinOp("+",BinOp("*",Int 2, Int 3), Int 5)
    ```

    Then all you have to do is call `interp p1` to test it.

3 Handing In

To turn in your program, commit the final version to the repository located at
`http://host220.cns.iit.edu/svn/cs440/`

Make sure your program will compile. If it doesn't, you will **not** get credit for it.

If you don't have time to finish for some reason, you get an *automatic* extension. You don't have to ask for it, and you will not be penalized for taking it. However, we will not grant further extensions once the assignment has entered the extension time unless it has been cleared in advance.